(2)

Technical Report

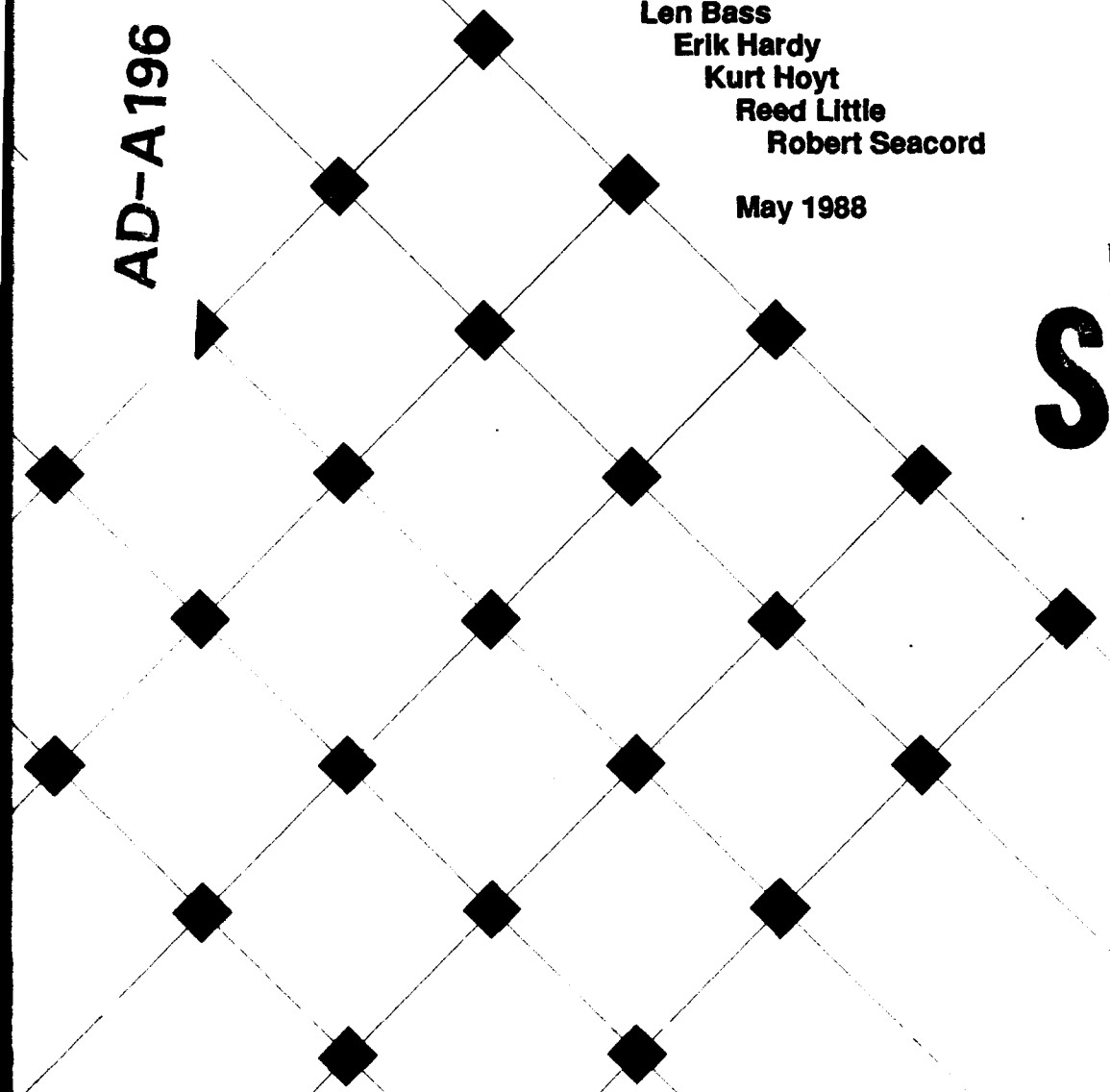
CMU/SEI/88-TR-6
ESD-TR-88-007

Carnegie-Mellon University
Software Engineering Institute

The Serpent Runtime Architecture and Dialogue Model

Len Bass
Erik Hardy
Kurt Hoyt
Reed Little
Robert Seacord

May 1988



Technical Report
CMU/SEI-88-TR-6
ESD-TR-88-007
May 1988

The Serpent Runtime Architecture and Dialogue Model

Len Bass
Erik Hardy
Kurt Hoyt
Reed Little
Robert Seacord

User Interface Prototyping Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

Serpent Runtime Architecture and Dialogue Model

Abstract: The separation of the user interface portion of a software system from the functional portion is intended to enable the production of tools to deal with the user interface, and to raise the quality and modularity of resulting software systems. One class of such separation tools that have been developed is the User Interface Management System (UIMS). This paper describes the runtime architecture and dialogue model of a particular UIMS named Serpent.

Serpent uses existing software systems to create a UIMS based on a structured production model to specify the dialogue, and uses a database approach for communication between its internal layers. The model for the dialogue in Serpent supports simultaneity of subdialogues and presents the dialogue specifier with a model that views data as mapping from the application to the presentation.

The database approach for communication between the layers provides a model that application programmers understand well and find easy to use. The approach also provides the power necessary to decouple the application structures from the structures implicit in the user interface.

1. Introduction

Highlighting the user interface of programs as a separate area of functionality needing special tools and special concepts has resulted in a new software architecture known as the User Interface Management Systems (UIMS). This architecture calls for dividing an application into layers: the functional core of the application, the dialogue control layer, and the presentation layer. The exact functionality resident in each layer is subject to dispute but, in general, the presentation layer is responsible for layout and device issues, the dialogue control defines the structure of the dialogue between the user and the application, and the application layer provides the functionality for the whole system.

In the last several years, a large number of UIMS have been built [1, 14, 15, 9, 10, 5] and the issues involved in the architecture have become clearer. Serpent (Software Engineering Rapid Prototyping EnvironmenT) is a UIMS under development. It takes a particular view of some of these issues. This paper discusses the runtime architecture of Serpent and the model used within Serpent to control the dialogue. In particular, Serpent:

- Uses the X Toolkit and window system for presentation and feedback. This results in a four-level model such as that suggested at the 1986 SIGGRAPH workshop [3].

- Takes advantage of a structured production model for dialogue specification that supports multithread dialogues and multiple views of the same data. The structuring imposed on the productions provides the interface designer with a model that views the interface as a mapping *from* the application data *to* the presentation objects. This is in contrast to models that view the application and the presentation as symmetric with respect to the dialogue. The model also groups presentation objects based on visibility conditions allowing collections of presentation objects to be treated uniformly by the logic of the dialogue.
- Uses a database approach to the interface between the layers. The database approach has the advantages of:
    - Ease of use for application programmers who use Serpent
    - Power to allow the decoupling of user interface structure from application structure

2. Serpent Architecture

The decomposition of Serpent reveals four layers: the application, the syntactic dialogue manager, the lexical dialogue manager, and the presentation layer. Figure 2-1 illustrates this decomposition.

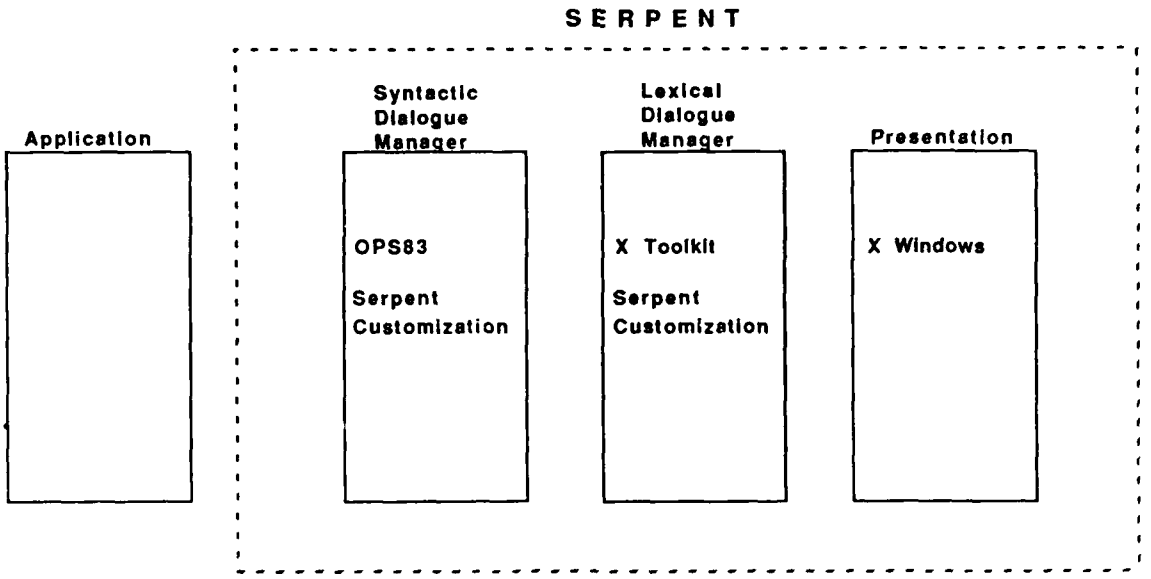


Figure 2-1: Serpent Decomposition

Serpent itself consists of the latter three layers, since the application is outside Serpent's domain. The terms "syntactic" and "lexical," when applied to the dialogue manager, are intended to suggest a matching with the Foley and Van Dam model [6] and are also intended to describe the power of the underlying language used to specify the actions of these layers. The three layers of Serpent use existing software systems heavily. The syntactic dialogue manager has OPS83 [8] at its core, the lexical dialogue manager has the X Toolkit [12] at its core, and the presentation layer is exclusively the X window system [17]. What is new about Serpent is the collection of these components into a coherent UIMS, the model presented to the dialogue specifier, and the use of database concepts for the communication among the layers.

The sections that follow describe briefly each of the three Serpent layers.

2.1. Presentation Layer

The X window system manages the resources associated with a bit-mapped workstation. This includes the windows, fonts, mouse cursors, and both mouse and keyboard input devices. X is structured as a server with various clients. The system assumes a rectangular overlapping model of window management and reports to its clients about low level events such as mouse movement and keyboard and button presses. The use of X for the window manager corresponds directly to the presentation layer in the Seeheim architecture [16] or the workstation agent of the Seattle conference [11].

2.2. Lexical Dialogue Manager

The Serpent lexical dialogue manager consists of the X Toolkit, as well as Serpent customizations to the toolkit. The customization provides the mapping between the syntactic and lexical dialogue managers, and some additions to the functionality furnished in the initial release of the X Toolkit. The general model followed in the use of X within the Serpent system is that the lexical actions are specified by the syntactic dialogue manager and performed by the lexical dialogue manager. The method of specification of the lexical actions is described below.

2.2.1. X Toolkit

The X Toolkit consists of a collection of "widgets" that handle policy matters and serve as the "interactors" of Serpent. The widgets in the X toolkit include command buttons, forms, dialogue boxes, etc. Widgets are implemented as X windows, which means that widgets are constrained to be rectangular. The widgets are parameterized so that the program that uses the widget can specify size, internal text or icon, and location. The term "application" program is applied to this program in the X Toolkit documentation. The term "widget driver" refers to the code that directly controls the widgets.

More important, from the perspective of Serpent, is that the widgets handle low-level feedback and the feedback actions are also parameterized. Widgets also generate events that are reported to the widget driver. Figure 2-2 shows how a command button responds to various actions of a mouse. The widget understands the basic events of "mouse entered, button down, button up" and has an action of "set". The X toolkit allows the actions taken in response to the basic events to be specified by the widget driver. A portion of the customization of the toolkit uses this ability for limited control of both low-level feedback and of the events reported to the syntactic dialogue manager.

The X Toolkit has base widgets (command button, text, Boolean button and scroll bar, for example) and composite widgets (button box, menu, and form, for example). Composite widgets are collections of base widgets bound together by a particular geometry manager.

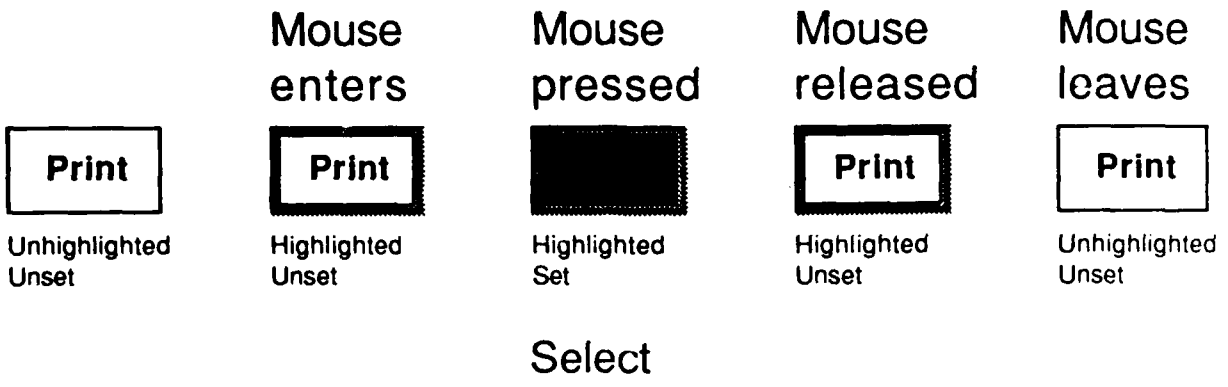


Figure 2-2: Command Button Widget

2.2.2. Serpent Customization of the X Toolkit

The customization done to the X toolkit provides two distinct functions. First, it acts as the interface to the toolkit, specifying that events are of interest and the parameters of the widgets. Second, a collection of graphic primitives has been implemented on top of X to allow interactions with objects that are not rectangles, such as lines and arcs. This bypasses the difficulty involved in the restrictions caused by rectangular widgets in the X toolkit for both line specification and movement.

The interface between lexical and the syntactic dialogue managers consists of two portions. The first portion is the description of the presentation and the second portion is the description of the interactions. The presentation is described in terms of widgets (both X toolkit widgets and enhancements). Each widget is described with a collection of specific values as presentation attributes. The presentation attributes are not only size, location, and contents but may include items associated with feedback, such as cursor shape or border size when the cursor is within a widget. The individual widgets do not refer to any other existing widgets except when they are components of a composite widget. In this case there is a reference to the parent of a particular widget.

The second portion of the interface consists of a definition of syntactic events in terms of the lexical events that the X toolkit understands. For example, suppose that in a drawing program there is a palette of shapes, a target window, and a single-button mouse. One interaction technique might be to place the cursor over a shape, press down on the button, and drag the shape to a desired place on the screen. Feedback might be a boundary box to show location. From a high-level perspective this is a "copy." The same set of interactions, when applied to a shape in the target window, might be a "move." The lexical events in this

example are button down, drag, display boundary box feedback, and button up. The syntactic events are move or copy. The syntactic dialogue manager passes a regular expression to the lexical dialogue manager. This expression defines a syntactic event in terms of lexical events. A regular expression is downloaded from the syntactic dialogue manager whenever the controlling dialogue wishes to define a new syntactic event.

Whenever a definition for a new event is downloaded to the lexical dialogue manager, it is checked for consistency with the other defined syntactic events. A certain sequence can have only one meaning at any point in time. If an inconsistent definition is added, an error condition is returned to the syntactic dialogue manager and no new event is defined.

Only the events returned by the X toolkit and certain feedback actions are specified by a regular expression. There is no attempt to specify geometry or presentation attributes through a regular expression. These matters are left for the syntactic dialogue manager to specify.

2.3. Syntactic Dialogue Manager

The syntactic dialogue manager controls the execution of the logic of the dialogue. It consists of OPS83 plus associated data management and communication functions. The dialogue is specified using the model described in this subsection. The dialogue is translated into OPS83 production rules that are executed by the syntactic dialogue manager. The functionality of the syntactic dialogue manager is described in much more detail in Section 6.

OPS83 acts as the runtime kernel of the Serpent syntactic dialogue managers. The rules for OPS83 are derived from the specification of the dialogue. Figure 2-3 shows how the specification is transformed into OPS83 rules. The dialogue specifier interacts with an editor to create a dialogue. This interaction is partially graphical and partially textual. The output of the interactive editing is a totally textual representation of the dialogue that, in turn, is compiled into OPS83 rules. These rules are compiled by the OPS83 compiler into an executable dialogue manager.

The resulting dialogue manager has the dialogue linked into it and is not an interpreter for the dialogue. Thus, rather than having one dialogue manager to interpret many different dialogues, there is one executable dialogue manager per dialogue.

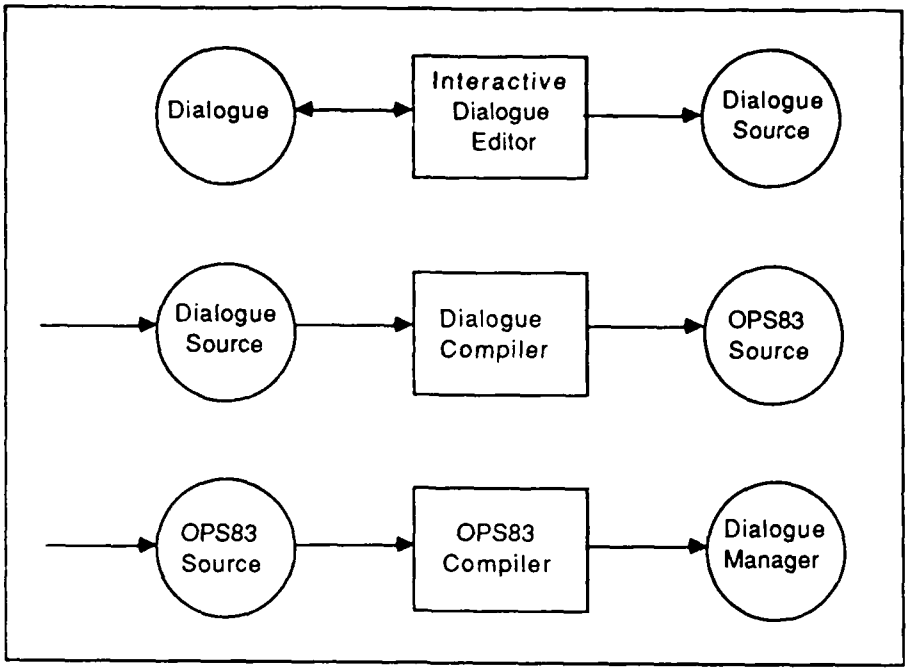


Figure 2-3: Specification Transformations

3. Data Flow

Figure 3-1 illustrates data flow within a system using Serpent.

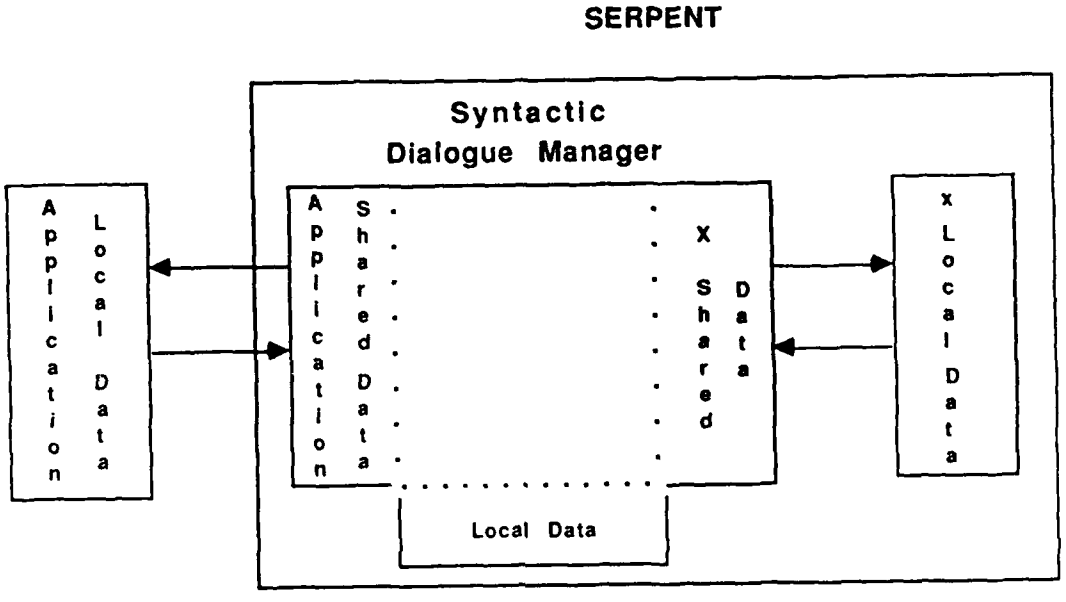


Figure 3-1: Serpent Data Flow

The application views Serpent as a system that manages the database of entities visible to the end user. The entities are viewed as relations, and the application requests Serpent to create a new tuple in a relation, modify an existing tuple, or delete an existing tuple. The database that Serpent manages for the application is called *application shared data*. The other side of this exchange is that the application is informed of end-user originated changes to the application shared data. The application shared data is *active* in that it informs clients of modifications to itself [7, 13].

The application shared data is managed by the syntactic dialogue manager. It manipulates the data according to the instructions in the dialogue and converts the data into presentation information placed in the X shared data. X shared data is another database managed by the syntactic dialogue manager; this database holds information about the X toolkit entities (widgets). The lexical dialogue manager modifies and is informed about modifications to the X shared data area in the same fashion as the application and the application shared data.

The flow of the data from an end user to the application is as follows:

1. End user moves mouse over command button. No data is transferred outside the X toolkit.
2. End user selects command button. X toolkit layer places identification of command button selected into interface.
3. Interface notifies dialogue manager of modifications to shared data.
4. Dialogue manager retrieves selection notification, transforms selection into application action, and places application action notification into application shared data.
5. Interface notifies application of modification to application shared data. Application retrieves modified data and acts upon it.

The syntactic dialogue manager maintains local data to control the logic of the dialogue and as staging between the various shared data areas.

4. Example

The example that follows illustrates Serpent details.

4.1. End User Functionality

The display in Figure 4-1 is drawn from a command and control application. This is the display that the end user of the example sees.

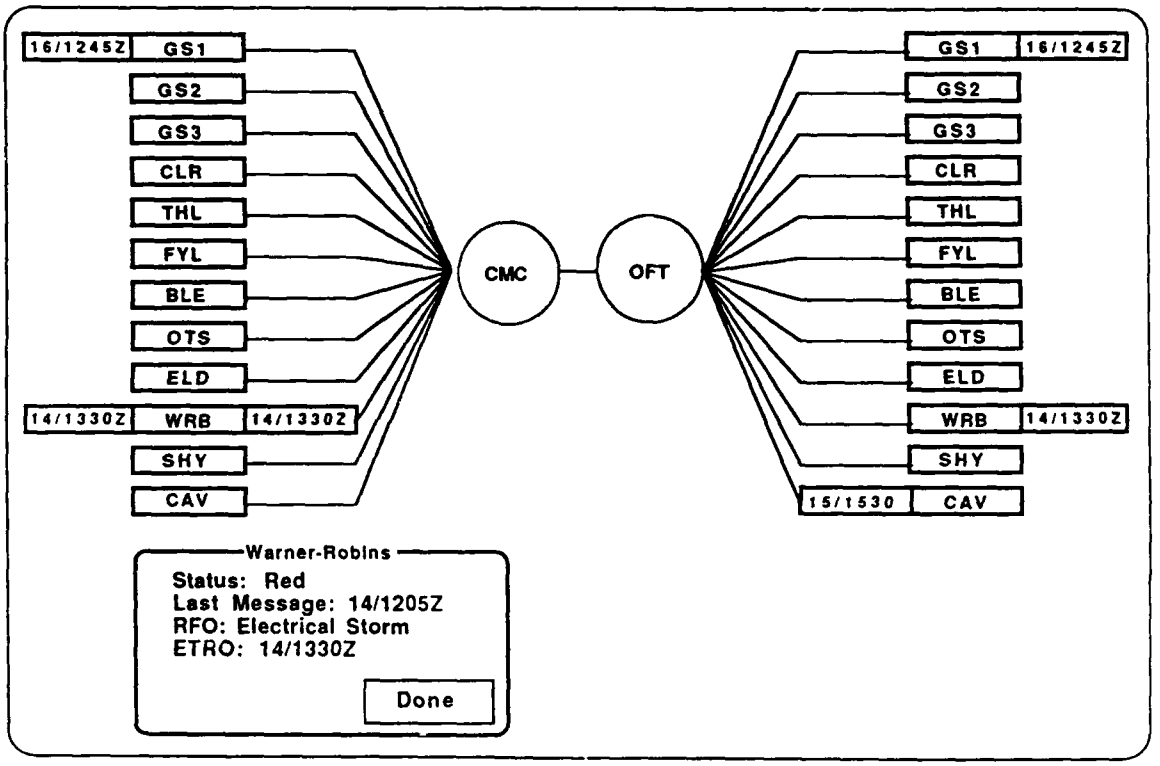


Figure 4-1: Application Example

The rectangular boxes on the right and left sides (e.g., GS1, GS2) represent sensor sites that detect information. The circles in the middle represent correlation centers where the information from all of the sensors is collected. Each sensor site sends its information to both correlation centers, which accounts for the repetition of the sensor site boxes on both the right and left sides of the display. The lines represent the communication path between a particular sensor site and a correlation center.

When a sensor site is determined to be non-operational, an *estimated time to return to operation* (ETRO) is displayed in association with the site. The ETRO is displayed in association with both occurrences of the sensor site. A particular communication line may not be operational, in which case the ETRO for that line is displayed over the line and next to the sensor site.

The end user may select one of the sensor sites and a detail window will appear giving more status information about the site. This detail window may be edited to modify the ETRO, the status, or the *reason for failure* (RFO). Figure 4-1 shows the result of selecting the WRB sensor.

Notice that the ETRO for a particular sensor site is always displayed twice. If the detail window for the sensor site has been selected, the ETRO is displayed three times. This notion that the same piece of information is displayed multiple times is called *multiple views* of data.

4.2. Application Functionality

An application treats Serpent as a database manager for the application shared data. When this concept is applied to the example, the functionality of the application consists of converting the information in the local database of the application into (and from) the application shared data area. Application shared data can be modified either by the application (in which case the modifications are of interest to Serpent) or by Serpent (in which case the application is explicitly informed of the modifications).

The example application maintains a database of sensor sites, communication lines, and correlation centers. Each component of the database has associated status information, for example, ETRO. It communicates this information to Serpent by writing the information to the application shared data.

The example application has two sources of information. Either the end user inputs information or information arrives through direct communications from another computer. A high-level outline of the application program's sequence is:

- Initialize connection with Serpent.
- Retrieve data from local database and put into shared data.
- Notify Serpent that data is available.
- Do until exit.
    - Wait for either input from Serpent or message from another computer.
    - If input from Serpent then:
        - Get updated information from shared data.
        - Verify information.
        - Place updated information into local database.

- If message from another site then:
    - Place new information into local database.
    - Place new information into shared data.
    - Notify Serpent that new information is available.

Notice how the application treats Serpent as an active database manager. It places application information (about sensor sites and communication lines) into shared data and is subsequently informed of changes to this information. The application is ignorant of the details of the display. The application is also ignorant about the user asking for more detailed information about a particular sensor. This is the essence of separating the user interface details from the application program. Notice also that the application is not informed of end user actions until the actions have been transformed into the form necessary for shared data, that is, until the data is ready for application action.

An issue in user interface design is whether a modification requested by an end user is immediately reflected on the display or whether the display reflects the modification only when the application acknowledges the modification. Serpent does not prejudge this issue. During the design process decisions are made as to when a modification is to be reflected on the display, and the application and the dialogue are written accordingly.

5. Serpent Shared Data

From an application perspective, data is sent to a database that Serpent maintains. Changes to this database are also reported back to the application. The database model of how applications interact with the database manager is well understood by application programmers. The application either creates, modifies or deletes data in the database. It communicates the specifics of its requests through a schema that defines what data is in the database.

The example application is described above as placing data into shared data. Application shared data is conceived as a database of information managed by Serpent and available to the end user. The database is relational (tabular) in structure. In terms more familiar to programmers, shared data is a collection of records. Each record is composed of scalar data types. In the example, one possible structure for some of the data is:

- Sensor site data table
    - site abbreviation
    - site status
    - site full name
    - last message
    - *rfo*
    - etro
- Communication line data table
    - from sensor site
    - to correlation center
    - status
    - etro

The terminology here depends on whether a programming or a database perspective is used. The elements in shared data are records (tables, relations). Each table in shared data is a collection of rows (tuples, instances of records) or a collection of columns (fields, attributes). When a new row is created (either by the application or by Serpent) it is given a unique identification (shared data element id). That is, the identification of a row is independent of the data stored in a row. Serpent assigns the primary keys for the database that Serpent manages, and the application manages any association between the primary key and particular data values.

This database view of shared data occurs not only between the application and Serpent but also within components of Serpent. The syntactic dialogue manager and the lexical dialogue manager share data through *X shared data*. This data consists of widgets or graphical objects and their attributes, which fits exactly into the database model. There is a data table for each widget type and its fields are the attributes of the widgets. For example, there is a command-button data table and its columns are attributes of the command button (size,

color, location). Each row in that table represents a different occurrence of a command button. An additional attribute exists in the command button table that gives the owner within the dialogue of an individual command button. Thus, independent menus composed of command buttons are easily managed. When the end user selects a command button its owner within the dialogue is notified if the selection. This owner knows that menu it is managing.

6. Model Used in the Syntactic Dialogue Manager

When two or more dialogues are running concurrently, the resulting dialogue is termed multithreaded. The support of multithreaded dialogues has become an important requirement for UIMS dialogue models. These two dialogues may be independent, such as when a user's focus of attention changes in the middle of a dialogue, or they may be intertwined, such as when a user uses two input devices simultaneously.

Production systems have been used as a means of dealing with the simultaneity involved in multithreaded dialogues [10, 5]. Serpent also uses a production system model to describe the dialogue, although, in the case of Serpent, the productions with that a dialogue is specified (view controllers) are more structured and at a higher level than the productions that are actually executed. View controllers are also nested, which gives the dialogue specifier the view that they map from application data to technology objects [2].

6.1. View Controllers

In Serpent, the actual dialogue between the end user and the application is executed in terms of *view controllers*. A view controller performs two main functions:

- Map specific data in the application shared data into objects on the display with which the end user can interact.
- Control, at a high level, the interactions that the end user has with those objects.

A dialogue is specified in terms of view controller *templates*. A template maintains a watch on application shared data for certain specific data conditions. A view controller is created when data that satisfies a watching view controller template is placed into application shared data.

The actual view controller has the following functions:

- Tie a particular tuple in shared data space to the view controller.
- Map that data into display objects visible to the end user.
- Perform actions when the end user interacts with the display objects.
- Maintain local information.

In general, a view controller template consists of four components:

1. Creation condition for new view controller
2. Actions on creation of new view controller
3. List of display objects, each object consisting of a collection of attributes for presentation and methods to respond to end user actions
4. Actions on destruction of created view controller

The subsections that follow describe these components in general and in terms of the previous view controller example.

6.1.1. View Controllers as Used in the Example

In the example display in Figure 4-1, there are two rectangles associated with every sensor site (one on the right of the screen and one on the left). For each sensor site, there is a specific tuple in the sensor site data table that has the information for the sensor site, and also a view controller that maps that tuple into the two rectangles. A separate view controller causes the detail window for a sensor site to be displayed when the end user selects the sensor site. The view controller that controls the selected sensor site rectangle is informed when the sensor site has been selected and causes the creation of the detail window view controller.

Even though there is a separate view controller for each sensor site, there is a single view controller template from that the view controllers are created. This view controller template specifies the condition under which a new view controller occurs. The created view controller then maps the particular tuple representing a sensor site into the rectangles on the display, interprets any selection of one of these rectangles as the signal to create the detail window view controller, and maintains some local information.

The view controller template for the sensor site is:

Sensor Site View Controller

- Creation condition: sensor site abbreviation is new
- Objects:
    - Left sensor site button (creates new tuple in X shared data command button data table)
        - attributes:
            - color
            - size
            - location
            - text in rectangle
        - method
            - select: Create view controller that brings up detail box.
    - Right sensor site button (creates new tuple in X shared data in command button data table)
        - attributes:
            - color
            - size
            - location

- text in rectangle

- method
  - select: Create view controller that brings up detail box.

6.1.1.1. Creation Condition

A view controller template waits until a specific condition is satisfied. In the example, the condition is the existence of a new sensor site in application shared data. Once that condition is satisfied, a view controller is created and performs its actions. The creation condition satisfies two purposes. First, it determines when a new view controller is created from a view controller template, and second, it associates a tuple from the shared data with the newly-created view controller. In the example, when a new sensor site abbreviation is placed into shared data, a view controller is constructed from the template and is associated with the tuple that contains the new sensor site abbreviation. If the view controller exists and the tuple with its particular sensor site abbreviation is deleted, then the view controller ceases to exist.

In general, a view controller creation condition may be any condition on the attributes in a single shared data table modified by any local information maintained within the dialogue. When the condition is satisfied by certain fields of a particular tuple in a table, then the value of all the fields in the tuple are bound to the view controller. These other values are typically used for construction of the attributes of objects.

6.1.1.2. Actions on Creation

When a view controller is created, then its actions on creation are executed. These actions can be any of the legal ways in that the dialogue can manipulate shared data, manipulate local information, or send information to the application.

Possible actions on creation for the previous example might be to increment a count of sensor sites or to initialize the flag used to control the display of the detail window.

6.1.1.3. Objects

Each view controller template describes a collection of objects that are created when a view controller is created from the template. These objects correspond to the shapes on the display and are bound to the newly created view controller. The objects have attributes that control how they are presented to the end user, and methods that determine the high-level interactions that the end user can have with the object. All of the objects within a particular view controller are created when the view controller is created and, thus, the view controller acts as a mechanism for grouping display objects based on visibility conditions.

In the previous sensor-site example, each view controller creates two objects, the right and left sensor rectangles. The attributes of these objects determine the size, location, color and internal text of the objects. The values of the attributes may depend upon the values of the fields in the sensor-site data table tuple that caused the view controller to be created.

In general, the values taken on by the attributes may depend upon values of the fields in a shared data table or from local information maintained by the dialogue. Several different values can enter into the calculation of a single attribute. In particular, the values can be drawn from the tuple with which the view controller is associated, from values local to the dialogue, or from attributes of other objects. The ability to reference attributes of other objects allows the line objects in Figure 4-1 to be specified such that they are connected to the sensor site objects. Thus, inter-object geometric relationships (excluding parent-child) are enforced by the dialogue (through the syntactic dialogue manager) and not at the lexical level. Parent-child geometric relationships (one widget is a component of another widget and the position of the child widget is specified relative to the position of the parent) are enforced at the lexical level.

The objects also have methods that determine their reaction to end-user actions. In the previous sensor-site example, selecting one of the sensor-site rectangles is the only action an end user can perform on the rectangles. The mechanism for the selection is managed by the X toolkit as specified previously in this paper. When a sensor site rectangle is selected, the lexical dialogue manager notifies the syntactic dialogue manager that a selection has occurred *for a particular object*. This object belongs to a particular view controller and, consequently, the particular tuple associated with that view controller is known. When a selection occurs the sensor site view controller arranges for a detail window view controller to be created. It could do this by setting a local flag that the detail window view controller template uses as its creation condition.

6.1.1.4. Actions at Destruction

When the creation condition of a view controller becomes false, the view controller is deleted from the system and its objects are removed from the display. It is also possible to specify other actions to be performed upon destruction. Possible actions for the previous sensor-site example might be to decrement a counter of sensor sites or to inform the application of certain information.

6.1.2. Nesting of View Controllers

One view controller template can be specified to be nested within another view controller template. This nesting carries through to the actual view controllers created from the templates. A nested view controller inherits the tuple that caused the creation of its predecessor. In the previous sensor-site example, the detail window view controller is nested within the sensor site view controller. Hence, when the detail window view controller is created, it inherits the tuple that caused the sensor site view controller to be created. This means that the detail window view controller presents certain information about a sensor site and the nesting insures that the information is associated with the correct sensor site.

6.2. Threads of Control Within Dialogues

A dialogue is a collection of view controller templates. Each of the view controller templates has a creation condition. The order in which the view controllers are created depends upon the data placed into shared data by the application and the actions of the end user. A sub-dialogue is a collection of view controllers that perform one particular task, for example, create display in Figure 4-1. It is possible within a dialogue to have multiple subdialogues, and there are no a priori timing constraints on the order in which those sub-dialogues are executed.

Actions of the dialogue are determined by the actions of the application and of the end user; it is possible to have multiple subdialogues active simultaneously. For example, Figure 4-1 may represent only one portion of a total display and the end user may select a sensor site, have the detail window displayed, leave it displayed and proceed with quite a different task in a different portion of the display. Within Serpent, view controllers are created and methods are used totally in response to end user and application actions. In particular, several subdialogues may be carried on in parallel. This allowance of simultaneity of subdialogues represents the power of the production model used in Serpent.

6.3. Multiple Views of Data Within Serpent

The data shared between the application and Serpent has one tuple for each collection of application data. The fact that a particular piece of data may be displayed multiple times on a display is reflected only in the dialogue and not in the shared data. In the previous sensor-site example, the ETRO for a particular sensor site may be displayed as many as three times. Since the view controllers manage the mapping from application data to presentation objects, Serpent is aware of which view controllers depend upon which values of shar data. Thus, when a particular piece of shared data is modified, Serpent is able to ensure consistency with all of the presentations of that particular piece of shared data.

6.4. User Model of the Data

The basis of the UIMS architecture is that there is a distinction between the information content of application data and the form in which the data is presented. In this section, the argument is made that it is the *structure* of application data that should be separated from the structure of the presentation. The user forms a model of the application being used, on the basis of a perceived structure of the information being presented. As long as the information content of the data remains the same, modifications to the actual structure of application data should be hidden from the end user. This mimics the distinction used in databases of conceptual level and external level [4].

The distinction between conceptual level and external level in databases allows the end user to have a different view of the structure of the data in the database than the structure

that actually exists. When this idea is translated to user interfaces, it means that there should be a distinction between the structure of the data that the end user sees and the structure of the data that the application manages. This emphasis on structure of data is independent of the form that the data is presented to the end user.

If Figure 4-1 is examined for information content, there are separate entities of sensor site and communication line, regardless of how these entities are displayed. The sensor-site example's application shared data mirrors these two entities in two separate data tables. Presumably the application has the same entities. Yet, if the separation between application functionality and user interface is to be truly achieved, it should be possible to have two entities in the user interface derived from one or three entities in the application. It should be possible to have a *database view* of the structure of the application data.

Suppose that, in the sensor-site example, there were only one data table in application shared data with the following structure:

Communication line data table

- from sensor site
- to correlation center
- sensor site abbreviation
- sensor site status
- sensor site full name
- sensor site last message
- sensor site rfo
- sensor site etro
- communication line status
- communication line etro

This structure is not normalized, in the database sense, because information for a sensor site is replicated within each tuple having a particular sensor site abbreviation. But this is a possible structure that the application may maintain for the data. By defining an appropriate dialogue it should be possible to map this structure into the display in Figure 4-1 and present the user with a view of the data that distinguishes between communication lines and sensor sites.

In Serpent this is possible. The sensor site view controller template creates a view controller when a new sensor site abbreviation occurs in application shared data. This view controller is bound to the tuple within which the abbreviation occurred. Whether this abbreviation occurs once or several times within an application shared data table is irrelevant to the creation condition and the binding. The view controller template creates only one view controller per abbreviation.

This use of a creation condition and the associated binding to a tuple allows Serpent to have a subset of what is meant by *view* in the database sense, and allows a decoupling of the

structure of the user interface from the structure of the application data. This is important in maintaining true separation between the application and the user interface.

6.5. Timing of Dialogue Actions

The syntactic dialogue manager monitors the application shared data area and local dialogue information, and creates view controllers when a view controller template-creation condition has been satisfied. Attributes of objects are also modified when the data that the attribute reflects is modified. It is possible to have multiple view controllers created (or attributes recomputed) in response to a single change in shared data. From the point of view of the dialogue specifier all of these actions are simultaneous.

Some of these simultaneous actions, however, may be inconsistent. Because of the power of the model used within Serpent, it is not possible to guarantee correctness of a dialogue. Serpent imposes an ordering on these actions so that they are repeatable and explainable, but correctness of a dialogue is the responsibility of the dialogue specifier.

7. Summary

Serpent is a UIMS that uses a structured production model to specify dialogue, and that uses a database approach to the interface between the layers of the UIMS.

The structuring of the productions allows:

- Grouping of display objects according to the logic of the dialogue. This allows the existence of an object to be visible from examination of the dialogue.
- A view of the dialogue as mapping from application data to presentation objects. This gives the dialogue specifier a focus for structuring a dialogue.
- The ability to nest productions and allow one production to inherit data from its parent.

The use of a database model for the interface allows:

- The actions of the productions to be grounded in particular data that can be used to then control the presentation attributes of the objects.
- A model for a UIMS that is well understood by application programmers.
- An approach to the problem of separating the structure of the data seen by the end user from the structure of the data managed by the application.
- The same code to manage the interface between the application and the dialogue manager, as between the dialogue manager and the X toolkit layer.

References

[1] Buxton, W., Lamb, M.R., Sherman, D., Smith, K.C.
Toward a Comprehensive User Interface Management System.
Computer Graphics 17(3), July, 1983.

[2] Coutaz, Joelle.
PAC, an Object Oriented Model for Dialog Design.
Human-Computer Interaction - INTERACT '87.
Elsevier Science Publishers B.V. (North-Holland), 1987.

[3] Dance, J.R., Granor, T.E., Hill, R.D.,Hudson,S.E., Meads, J.,Myers, B.A., Schulert, A.
The Run-time Structure of UIMS-Supported Applications.
Computer Graphics 21(2), April, 1987.

[4] Date, C.J.
An Introduction to Database Systems, Vol I.
Addison Wesley, 1985.

[5] Flecchia, M.A. and Bergeron, R.D.
Specifying Complex Dialogs in Algae.
In *Proceedings SIGCHI 87.* 1987.

[6] Foley, J.D. and Van Dam, A.
Fundamentals of Computer Graphics.
Addison Wesley, 1982.

[7] Foley, J.D. and McMath, C.F.
Dynamic Process Visualization.
IEEE Computer Graphics and Applications 6(3), March, 1986.

[8] Forgy, C.L.
The OPS83 Report.
Technical Report CMU-CS-84-113, Carnegie Mellon University, Computer Science, 1984.

[9] Green, M.
University of Alberta User Interface Management System.
Computer Graphics 19(3), July, 1985.

[10] Hill, R.D.
Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues.
In *Proceedings SIGCHI 87.* 1987.

[11] Lantz, K.A., Tanner, P.P., Binding, C., Huang, K-T.,Dwelly, A.
Reference Models, Window Systems, and Concurrency.
Computer Graphics 21(2), April, 1987.

[12] McCormack, J., Asente, P., Swick, Ralph R.
X Toolkit Intrinsics.
Massachusetts Institute of Technology, 1987.

[13] Myers, B.A.
Creating Dynamic Interaction Techniques by Demonstration.
In *Proceedings SIGCHI 87*. 1987.

[14] Olsen, D.R. and Dempsey, E.P.
SYNGRAPH: A Graphic User Interface Generator.
Computer Graphics 17(3), July, 1983.

[15] Olsen, D.R., Jr.
MIKE: The Menu Interaction Kontrol Environment.
ACM Transactions on Graphics 5(4), October, 1986.

[16] Pfaff, G. (editor).
User Interface Management Systems.
Springer-Verlag, Berlin, 1985.

[17] Scheifler, R.W. and Gettys, J.
The X Window System.
ACM Transactions on Computer Graphics 5(2), April, 1986.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-6	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88-007
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: — PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) THE SERPENT RUNTIME ARCHITECTURE AND DIALOGUE MODEL	
12. PERSONAL AUTHOR(S) Bass, Len, Hardy, Erik, Hoyt, Kurt, Little, Reed, Seacord, Robert	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) May 1988 15. PAGE COUNT 34
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Dialogue managers, UIMS, Display design, user interface, prototyping systems, x toolkit

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The separation of the user interface portion of a software system from the functional portion is intended to enable the production of tools to deal with the user interface, and to raise the quality and modularity of resulting software systems. One calss of such separation tools that have been developed is the User Interface Management System (UIMS). This paper describes the runtime architecture and dialogue model of a particular UIMS named Serpent.

Serpent uses existing software systems to create a UIMS based on a structured production model to specify the dialogue, and that uses a database approach for communication between its ingternal layers. The model for the dialogue in Serpent supports simultaneity of subdialogues and presents the dialogue specifier with a model that views data as mapping from the application to the presentation.

The database approach for communication between the layers provides a model that application programmers understand well and find easy to use. The approach also provides the power necessary to decouple the application structuresfrom the structures implicit in the user interface.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473 83 APR EDITION OF 1 JAN 73 IS OBSOLETE